Data Entered)

ION PAGE

READ INSTRUCTIONS BEFORE COMPLETEING FORM

AD-A204 506

12. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

Ada Compiler Validation Summary Report: Naval Underwater Systems Center, ADAVAX, Version 1.7 w/ OPT VAX 8600 (Host) to VAX 8600 (Target)

5. TYPE OF REPORT & PERIOD COVERED
16 June 1988 to 16 June 1988

6. PERFORMING ORG. REPORT NUMBER
-

7. AUTHOR(s)
National Bureau of Standards
Gaithersburg, MD

8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION AND ADDRESS
National Bureau of Standards
Gaithersburg, MD

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
Ada Joint Program Office
United States Department of Defense
Washington, DC 20301-3081

12. REPORT DATE

13. NUMBER OF PAGES

14. MONITORING AGENCY NAME & ADDRESS(If different from Controlling Office)
National Bureau of Standards
Gaithersburg, MD

15. SECURITY CLASS (of this report)
UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20 if different from Report)

UNCLASSIFIED

DTIC ELECTE FEB 07 1989 S D H

18. SUPPLEMENTARY NOTES

19. KEYWORDS (Continue on reverse side if necessary and identify by block number)

Ada Programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, ACVC, Validation Testing, Ada Validation Office, AVO, Ada Validation Facility, AVF, ANSI/MIL-STD-1815A, Ada Joint Program Office, AJPO

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

ADAVAX, Version 1.7 w/OPT, Naval Underwater System Center, National Bureau of Standards, VAX 8600 under VAX/VMS, Version 4.5 (Host) to VAX 8600 under VAX/VMS Version 4.5 (Target), ACVC 1.9

DD FORM 1473 1 JAN 73 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-LF-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

89 2 6 051

DTIC FILE COPY

AVF Control Number: NBS88VUSN525_2

Ada Compiler
VALIDATION SUMMARY REPORT:
Certificate Number: 880616S1.09146
NAVAL UNDERWATER SYSTEMS CENTER
ADAVAX, VERSION 1.7 w/ OPT
VAX 8600 AND VAX 8600

Completion of On-Site Testing:
16 JUN 88

Prepared By:
Software Standards Validation Group
Institute for Computer Sciences and Technology
National Bureau of Standards
Building 225, Room A266
Gaithersburg, Maryland 20899

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington, D.C. 20301-3081

Ada Compiler Validation Summary Report:

Compiler Name: ADAVAX, VERSION 1.7 w/ OPT

Certificate Number: 880616S1.09146

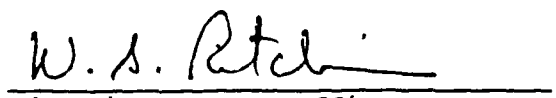
Host:	Target:
VAX 8600 under VAX/VMS, Version 4.5	VAX 8600 under VAX/VMS, Version 4.5

Testing Completed 16 Jun 88, Using ACVC 1.9

This report has been reviewed and is approved.

Ada Validation Facility
Dr. David K. Jefferson
Chief, Information Systems
Engineering Division
National Bureau of Standards
Gaithersburg, MD 20899

Ada Validation Organization
Dr. John F. Kramer
Institute for Defense Analyses
Alexandria, VA 22311



Ada Joint Program Office
Virginia L. Castor
Director
Department of Defense
Washington DC 20301

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies—for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

This information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation dependent but permitted by the Ada Standard. Six classes of test are used. These tests are designed to perform checks at compile time, at link time, and during execution.

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard

To attempt to identify any unsupported language constructs required by the Ada Standard

To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted by the National Bureau of Standards according to policies and procedures established by the Ada Validation Organization (AVO). On-site testing was completed 16 Jun 88 at NUSC, Newport, RI.

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

Ada Information Clearinghouse
Ada Joint Program Office
OUSDRE
The Pentagon, Rm 3D-139 (Fern Street)
Washington DC 20301-3081

or from:

Software Standards Validation Group
Institute for Computer Sciences and Technology
National Bureau of Standards
Building 225, Room A266
Gaithersburg, Maryland 20899

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.3 REFERENCES

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

2. Ada Compiler Validation Procedures and Guidelines. Ada Joint Program Office, 1 January 1987.

3. Ada Compiler Validation Capability Implementers' Guide., December 1986.

1.4 DEFINITION OF TERMS

ACVC	The Ada Compiler Validation Capability. The set of Ada programs that tests the conformity of an Ada compiler to the Ada programming language.
Ada Commentary	An Ada Commentary contains all information relevant to the point addressed by a comment on the Ada Standard. These comments are given a unique identification number having the form AI-ddddd.
Ada Standard	ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
Applicant	The agency requesting validation.
AVF	The Ada Validation Facility. The AVF is responsible for conducting compiler validations according to procedures contained in the Ada Compiler Validation Procedures and Guidelines.
AVO	The Ada Validation Organization. The AVO has oversight authority over all AVF practices for the purpose of maintaining a uniform process for validation of Ada compilers. The AVO provides administrative and technical support for Ada validations to ensure consistent practices.

Compiler	A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.
Failed test	An ACVC test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.
Host	The computer on which the compiler resides.
Inapplicable test	An ACVC test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.
Language Maintenance	The Language Maintenance Panel (LMP) is a committee established by the Ada Board to recommend interpretations and Panel possible changes to the ANSI/MIL-STD for Ada.
Passed test	An ACVC test for which a compiler generates the expected result.
Target	The computer for which a compiler generates code.
Test	An Ada program that checks a compiler's conformity regarding a particular feature or a combination of features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.
Withdrawn test	An ACVC test found to be incorrect and not used to check conformity to the Ada Standard. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce compilation or link errors.

Class A tests check that legal Ada programs can be successfully compiled and executed. There are no explicit program components in a Class A

test to check semantics. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntax or semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

Class C tests check that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters—for example, the number of identifiers permitted in a compilation or the number of units in a library—a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time—that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated.

Two library units, the package REPORT and the procedure CHECK FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK FILE is used to check the contents of text files written by some of the Class C tests for chapter 14 of the Ada Standard. The operation of

REPORT and CHECK_FILE is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of the tests in the ACVC follow conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and place features that may not be supported by all implementations in separate tests. However, some tests contain values that require the test to be customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of validation are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: ADAVAX, VERSION 1.7 w/ OPT

ACVC Version: 1.9

Certificate Number: 880616S1.09146

Host Computer:

Machine:	VAX 8600
Operating System:	VAX/VMS VERSION 4.5
Memory Size:	32 MB

Target Computer:

Machine:	VAX 8600
Operating System:	VAX/VMS VERSION 4.5
Memory Size:	32 MB

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. The tests demonstrate the following characteristics:

- Capacities.

  The compiler correctly processes tests containing loop statements nested to 65 levels, block statements nested to 65 levels, and recursive procedures separately compiled as subunits nested to 17 levels. It correctly processes a compilation containing 723 variables in the same declarative part. (See test D55A03A..H (8 tests), D56001B, D64005E..G (3 tests), and D29002K.)

- Universal integer calculations.

  An implementation is allowed to reject universal integer calculations having values that exceed SYSTEM.MAX_INT. This implementation processes 64 bit integer calculations. (See tests D4A002A, D4A002B, D4A004A, and D4A004B.)

- Predefined types.

  This implementation supports the additional predefined types LONG_INTEGER and LONG_FLOAT in the package STANDARD. (See tests B86001BC and B86001D.)

- Based literals.

  An implementation is allowed to reject a based literal with a value exceeding SYSTEM.MAX_INT during compilation, or it may raise NUMERIC_ERROR or CONSTRAINT_ERROR during execution. This implementation raises NUMERIC_ERROR during execution. (See test E24101A.)

- Expression evaluation.

  Apparently some default initialization expressions or record components are evaluated before any value is checked to belong to a component's subtype. (See test C32117A.)

Assignments for subtypes are performed with the same precision as the base type. (See test C35712B.)

This implementation uses no extra bits for extra precision. This implementation uses all extra bits for extra range. (See test C35903A.)

Sometimes NUMERIC_ERROR is raised when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

Apparently NUMERIC_ERROR is raised when a literal operand in a fixed-point comparison or membership test is outside the range of the base type. (See test C45252A.)

Apparently underflow is not gradual. (See tests C45524A..Z.)

- Rounding.

The method used for rounding to integer is apparently round away from zero (See tests C46012A..Z.)

The method used for rounding to longest integer is apparently round away from zero (See tests C46012A..Z.)

The method used for rounding to integer in static universal real expressions is apparently round toward zero (See test C4A014A.)

- Array types.

An implementation is allowed to raise NUMERIC_ERROR or CONSTRAINT_ERROR for an array having a 'LENGTH that exceeds STANDARD.INTEGER'LAST and/or SYSTEM.MAX_INT. For this implementation:

Declaration of an array type or subtype declaration with more than SYSTEM.MAX_INT components raises NUMERIC_ERROR (See test C36003A.)

NUMERIC_ERROR is raised when an array type with INTEGER'LAST + 2 components is declared.) (See test C36202A.)

NUMERIC_ERROR is raised when an array type with SYSTEM.MAX_INT + 2 components is declared. (See test C36202B.)

A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises no exception. (See test C52103X.)

A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises CONSTRAINT_ERROR when the length

of a dimension is calculated and exceeds INTEGER'LAST. (See test C52104Y.)

A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises no exception. (See test E52103Y.)

In assigning one-dimensional array types, the expression appears to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. In assigning two-dimensional array types, the expression does not appear to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

- Discriminated types.

During compilation, an implementation is allowed to either accept or reject an incomplete type with discriminants that is used in an access type definition with a compatible discriminant constraint. This implementation accepts such subtype indications. (See test E38104A.)

In assigning record types with disciminants, the expression appears to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

- Aggregates.

In the evaluation of a multi-dimensional aggregate, all choices appear to be evaluated before checking against the index type. (See tests C43207A and C43207B.)

In the evaluation of an aggregate containing subaggregates, all choices are evaluated before being checked for identical bounds. (See test E43212B.)

Not all choices are evaluated before CONSTRAINT_ERROR is raised if a bound in a nonnull range of a nonnull aggregate does not belong to an index subtype. (See test E43211B.)

- Representation clauses.

An implementation might legitimately place restrictions on

representation clauses used by some of the tests. If a representation clause is not supported, then the implementation must reject it.

Enumeration representation clauses containing noncontiguous values for enumeration types other than character and boolean types are supported. (See tests C35502I..J, C35502M..N, and A39005F.)

Enumeration representation clauses containing noncontiguous values for character types are supported. (See tests C35507I..J, C35507M..N, and C55B16A.)

Enumeration representation clauses for boolean types containing representational values other than (FALSE => 0, TRUE => 1) are supported. (See tests C35508I..J and C35508M..N.)

Length clauses with SIZE specifications for enumeration types are supported. (See test A39005B.)

Length clauses with STORAGE_SIZE specifications for access types are supported. (See tests A39005C and C87B62B.)

Length clauses with STORAGE_SIZE specifications for task types are supported. (See tests A39005D and C87B62D.)

Length clauses with SMALL specifications are supported. (See tests A39005E and C87B62C.)

Record representation clauses are not supported. (See test A39005G.)

Length clauses with SIZE specifications for derived integer types are supported. (See test C87B62A.)

- Pragmas.

The pragma INLINE is supported for procedures. The pragma INLINE is supported for functions. (See tests LA3004A, LA3004B, EA3004C, EA3004D, CA3004E, and CA3004F.)

- Input/output.

The package SEQUENTIAL_IO cannot be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101C, EE2201D, and EE2201E.)

The package DIRECT_IO cannot be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101H, EE2401D, and EE2401G.)

Modes IN_FILE and OUT_FILE are supported for SEQUENTIAL_IO. (See tests CE2102D and CE2102E.)

Modes IN_FILE, OUT_FILE, and INOUT_FILE are supported for DIRECT_IO. (See tests CE2102F, CE2102I, and CE2102J.)

RESET and DELETE are supported for SEQUENTIAL_IO and DIRECT_IO. (See tests CE2102G and CE2102K.)

Dynamic creation and deletion of files are supported for SEQUENTIAL_IO and DIRECT_IO. (See tests CE2106A and CE2106B.)

Overwriting to a sequential file truncates the file to last element written. (See test CE2208B.)

An existing text file can be opened in OUT_FILE mode, can be created in OUT_FILE mode, and cannot be created in IN_FILE mode. (See test EE3102C.)

More than one internal file can be associated with each external file for text I/O for reading only. (See tests CE3111A..E (5 tests), CE2110B, and CE2111D.)

More than one internal file can be associated with each external file for sequential I/O for reading only. (See tests CE2107A..D (4 tests), CE2110B, and CE2111D.)

More than one internal file can be associated with each external file for direct I/O for reading only. (See tests CE2107F..I (4 tests), CE2110B, and CE2111H.)

Temporary sequential files are given names. Temporary direct files are given names. Temporary files given names are not deleted when they are closed. (See tests CE2108A and CE2108C.)

- Generics.

Generic subprogram declarations and bodies can be compiled in separate compilations. (See tests CA1012A and CA2009F.)

Generic package declarations and bodies can be compiled in separate compilations. (See tests CA2009C, BC3204C, and BC3205D.)

Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.9 of the ACVC comprises 3122 tests. When this compiler was validated, 28 tests had been withdrawn because of test errors. The AVF determined that 340 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing. Modifications to the code, processing, or grading for 24 tests were required to successfully demonstrate the test objective. (See section 3.6.)

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	106	1048	1524	17	13	46	2754
Inapplicable	4	3	329	0	4	0	340
Withdrawn	3	2	21	0	2	0	28
TOTAL	113	1053	1874	17	19	46	3122

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER													TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14	
Passed	184	468	487	245	165	98	141	326	137	36	234	3	230	2754
Inapplicable	20	104	187	3	0	0	2	1	0	0	0	0	23	340
Withdrawn	2	14	3	0	1	1	2	0	0	0	2	1	2	28
TOTAL	206	586	677	248	166	99	145	327	137	36	236	4	255	3122

3.4 WITHDRAWN TESTS

The following 28 tests were withdrawn from ACVC Version 1.9 at the time of this validation:

B28003A	E28005C	C34004A	C35502P	A35902C	C35904A
C35904B	C35A03E	C35A03R	C37213H	C37213J	C37215C
C37215E	C37215G	C37215H	C38102C	C41402A	C45332A
C45614C	E66001D	A74106C	C85018B	C87B04B	CC1311B
BC3105A	AD1A01A	CE2401H	CE3208A		

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 340 test were inapplicable for the reasons indicated:

C35702A uses SHORT_FLOAT which is not supported by this implementation.

A35801E At the CASE statement (lines 54-63), the optimizer tries to identify which of the cases will be done during execution. The optimizer recognizes that the variable "I" which is of type integer, is not initialized and appropriately raises a PROGRAM_ERROR exception. NOTE: This test passes without the /OPTIMIZE option.

A39005G uses a record representation clause which is not supported by this compiler.

The following (14) tests use SHORT_INTEGER, which is not supported by this compiler.

C45231B	C45304B	C45502B	C45503B	C45504B
C45504E	C45611B	C45613B	C45614B	C45631B
C45632B	B52004E	C55B07B	B55B09D	

C45231D requires a macro substitution for any predefined numeric types other than INTEGER, SHORT_INTEGER, LONG_INTEGER, FLOAT, SHORT_FLOAT, and LONG_FLOAT. This compiler does not support any such types.

C45304A, C45304C and C46014A Optimization removes dead assignments leaving nothing and raising exception. NOTE: This test passes without the /OPTIMIZE option.

C45531M, C45531N, C45532M, and C45532N use fine 48-bit fixed-point base types which are not supported by this compiler.

C45531O, C45531P, C45532O, and C45532P use coarse 48-bit fixed-point base types which are not supported by this compiler.

B86001D requires a predefined numeric type other than those defined by the Ada language in package STANDARD. There is no such type for this implementation.

C86001F redefines package SYSTEM, but TEXT_IO is made obsolete by this new definition in this implementation and the test cannot be executed since the package REPORT is dependent on the package TEXT_IO.

C96005B requires the range of type DURATION to be different from those of its base type; in this implementation they are the same.

AE2101C, EE2201D, and EE2201E use instantiations of package SEQUENTIAL_IO with unconstrained array types and record types having discriminants without defaults. These instantiations are rejected by this compiler.

AE2101H, EE2401D, and EE2401G use instantiations of package DIRECT_IO with unconstrained array types and record types having discriminants without defaults. These instantiations are rejected by this compiler.

CE2105A, CE2107B..E (4 tests), CE2107G..I (3 tests), CE2110B, CE2111D, CE2111H, CE3109A, CE3111B..E (4 tests), and CE3114B are inapplicable because multiple internal files cannot be associated with the same external file for reading and writing. The proper exception is raised when multiple access is attempted.

The following 285 tests require a floating-point accuracy that exceeds the maximum of 9 digits supported by this implementation:

C24113F..Y (20 tests)	C35705F..Y (20 tests)
C35706F..Y (20 tests)	C35707F..Y (20 tests)
C35708F..Y (20 tests)	C35802F..Z (21 tests)
C45241F..Y (20 tests)	C45321F..Y (20 tests)
C45421F..Y (20 tests)	C45521F..Z (21 tests)
C45524F..Z (21 tests)	C45621F..Z (21 tests)
C45641F..Y (20 tests)	C46012F..Z (21 tests)

3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation behavior. Modifications are made by the AVF in cases where legitimate implementation behavior prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting a Class B test into sub-tests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behavior that wasn't anticipated by the test (such as raising one exception instead of another).

Modifications were required for 24 Class B tests.

The following Class B tests were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:

B2A003A	B2A003B	B2A003C	B33201C	B33202C
B33203C	B33301A	B37106A	B37201A	B37301I
B37307B	B38001C	B38003A	B38003B	B38009A
B38009B	B44001A	B51001A	B54A01C	B54A01L
B95063A	BC1008A	BC1201L	BC3013A	

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.9 produced by the ADAVAX was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of the ADAVAX using ACVC Version 1.9 was conducted on-site by a

validation team from the AVF. The configuration consisted of a VAX 8600 operating under VAX/VMS, VERSION 4.5 and a VAX 8600 target operating under VAX/VMS. VERSION 4.5. A tape containing all tests was taken on site by the validation team for processing. Tests that make use of implementation-specific values were customized on-site after the tape was loaded. Tests requiring modifications during the prevalidation testing were not included in their modified form on the tape. The contents of the tape were loaded directly onto the host computer.

After the test files were loaded to disk, the full set of tests was compiled and linked on the VAX 8600 and all executable tests were linked and run on the VAX 8600. Object files were linked and executed on the target. Results were printed from the target computer.

The compiler was tested using command scripts provided by SofTech, Inc. and reviewed by the validtion team. The compiler was tested using all default option settings except for the optimization option which was used.

Tests were compiled, linked, and executed (as appropriate) using a single host computer and a single target computer. Test output, compilation listings, and job logs were captured on tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

3.7.3 Test Site

Testing was conducted at NUSC, Newport, RI and was completed on 16 Jun 88.

APPENDIX A

DECLARATION OF CONFORMANCE

APPENDIX A

DECLARATION OF CONFORMANCE

Compiler Implementer: SofTech Inc.
460 Totten Pond Road
Waltham, MA 02254-9197

Ada Validation Facility: National Bureau of Standards (NBS)
Institute for Computer Sciences and Technology (ICST)
Software Standards Validation Group
Building 225, Room A266
Gaithersburg, MD 20899-9999

Ada Compiler Validation Capability (ACVC) Version: 1.9

BASE CONFIGURATION(S)

Base Compiler Name:	AdaVAX	Version: 1.7
Host Architecture -	ISA: VAX 8600	OS&VER #: VAX/VMS 4.5
Target Architecture -	ISA: VAX 8600	OS&VER #: VAX/VMS 4.5

Implementer's Declaration

I, the undersigned, representing SofTech, Inc., have implemented no deliberate extensions to the Ada Language Standard ANSI/MIL-STD-1815A in the compiler(s) listed in this declaration. I declare that the SofTech Inc. is the owner on record of the Ada language compiler(s) listed above and, as such, is responsible for maintaining said compiler(s) in conformance to ANSI-MIL-STD-1815A. All certificates and registrations for Ada language compiler(s) listed in this declaration shall be made only in the owner's corporate name.

Charles E. Coflin,
General Manager, Ada Engineering Division

April 4, 1988
Date

Owner's Declaration

I, the undersigned, representing U. S. Navy, take full responsibility for implementation and maintenance of the Ada compiler(s) listed above, and agree to the public disclosure of the final Validation Summary Report. I further agree to continue to comply with the Ada trademark policy, as defined by the Ada Joint Program Office. I declare that all of the Ada language compilers listed, and their host/target performance are in compliance with the Ada Language Standard ANSI/MIL-STD-1815A. I have reviewed the Validation Summary Report for the compilers(s) and concur with the contents.

William L. Wilder,
Ada/APSE Development Branch Head

April 14 1988
Date

APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of the VAX 8600, Version 4.5, are described in the following sections which discuss topics in Appendix F of the Ada Standard. Implementation- specific portions of the package STANDARD are also included in this appendix.

```
package STANDARD is


    type INTEGER is range -32_768 .. 32_767;

    type LONG_INTEGER is range -2_147_483_648 .. 2_147_483_647;

    type FLOAT is digits 6 range
                  2#0.1111_1111_1111_1111_1111_1111#E127 ..
                  2#0.1111_1111_1111_1111_1111_1111#E127;

    type LONG_FLOAT is digits 9 range -2#0.(14)1111#E127 ..
                                       2#0.(14)1111#E127;

    type DURATION is delta 2.0**(-14) range -131_072.0..
                                       131_072.0-2.0**(-14);


end STANDARD;
```

APPENDIX F

APPENDIX F OF THE ADA LRM FOR THE ADAVAX TOOLSET

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machinedependent conventions as mentioned in chapter 13 of MIL-STD-1815A, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics are described in the following sections which discuss topics one through eight as stated in Appendix F of the Ada Language Reference Manual (ANSI/MIL-STD-1815A). Two other sections, package STANDARD and file naming conventions, are also included in this appendix.

vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv

(1) Implementation-Dependent Pragmas

^^^

+++

This section may be copied from the applicant's documentation, but make sure it covers all the items below.

+++

The pragmas described below are implementation-defined.

Pragma TITLE (arg);

This is a listing control pragma. "Arg" is a CHARACTER string literal that is to appear on the second line of each page of every listing produced for a compilation unit in the compilation. At most, one such pragma may appear for any compilation, and it must be the first unit in the compilation (comments and other pragmas excepted).

pragma PAGE;

This is a listing control pragma. Using the pragma causes a page break in the listing; all code that follows pragma PAGE will start on a new page. The pragma may be placed anywhere inside a compilation unit.

The following notes specify the language_required definitions of the predefined pragmas. Unmentioned pragmas are implemented as defined in the Ada Language Reference Manual.

pragma INLINE (;subprogram_name);

There are three instances in which the INLINE pragma is ignored. Each of these cases produces a warning message which states that the INLINE did not occure.

a. If a call to an INLINEd subprogram is compiled before the actual body of the subprogram has been compiled (a routine call is made instead).

b. If the INLINEd subprograms compilation unit depends on the compilation unit of its caller (a routine call is made instead).

c. If an immediately recursive subprogram call is made within the body of the INLINEd subprogram (the pragma INLINE is ignored entirely).

pragma INTERFACE (language_name, subprogram_name);

Two language_names will be recognized and implemented:

ASMVAX_JSB, and ASMVAX_CALLS.

The language_name ASMVAX_JSB indicates that a subprogram written in the VAX/VMS assembler language will be called with a JSB instruction and the parameters passed in registers. The language_name ASMVAX_CALLS will provide interface to a VAX assembler language subprogram via the CALLS instructions, with the parameters passed on the stack, with the same parameter passing convertions used for calling Ada subprograms.

The users must ensure that an assembly-language body container for this specification will exist in the program library before linking.

pragma OPTIMIZE (arg)

This pragma is effective only when the "OPTIMIZE" option has been given to the compiler. The argument is either TIME or SPACE. If TIME is specified, the optimizer concentrates on optimizing code execution time. If SPACE is specified, the optimizer concentrates on optimizing code size.

pragma PRIORITY (arg)

The PRIORITY argument is an integer static expression value of predefined integer subtype PRIORITY. The pragma

has no effect in a location other than a task (type) specification or outer-most declarative part of a subprogram. If the pragma appears in the declarative part of a subprogram, it has no effect unless that subprogram is designated as the "main" subprogram at link time.

pragma SUPPRESS (arg[,arg])

Pragmas to suppress OVERFLOW_CHECK will have no effect for operations of integer types.

A SUPPRESS pragma will have effect only within the compilation unit in which it appears, except that a SUPPRESS of ELABORATION_CHECK applied at the declaration of a subprogram or task unit will apply to all calls or activations.

The pragmas MEMORY_SIZE, STORAGE_UNIT and SYSTEM_NAME are described in Section 13.7 of the Ada Language Reference Manual. They may appear only at the start of thefirst compilation when creating a new program library. In the ALS/N, however, since program libraries are created by the Program Library Manager and not by the compiler, the use of these pragmas is obviated. If they appear anywhere, a diagnostic of severity level WARNING is generated.

(2) Implementation-Dependent Attributes

There is one implementation-defined attribute in addition to the predefined attributes found in Apendix A of the Ada Reference Manual.

X'DISP

A value of type UNIVERSAL_INTEGER which corresponds to the displacement that is used to address the first storage unit occupied by a data object X at a static offset within an implemented activation record.

This attriabute differs from the ADDRESS attribute in that ADDRESS supplies the absolute address while DISP supplies the displacement relative to some base value (such as a stack frame pointer). It is the user's responsibility to determine the base value relevant to the attribute.

The following notes augment the language_required definitions of the predefined attributes found in Appendix A of the Ada Reference Manual.

T'MACHINE_ROUNDS	is false.
T'MACHINE_RADIX	is 2.
T'MACHINE_MANTISSA	if the size of the base type T is 32, MACHINE_MANTISSA is 24. if the size of the base type T is 64, MACHINE_MANTISSA is 56.
T'MACHINE_EMAX	is 127.
T'MACHINE_EMIN	is -127.
T'MACHINE_OVERFLOWS	is true.

(3) Package SYSTEM

The specification for the package SYSTEM is

```
package SYSTEM is

   -- Within the various implementations, no cooresponding package bodies
   -- are required for either the package STANDARD or the package SYSTEM.

   type ADDRESS is new LONG_INTEGER;
   type NAME    is (VAX_VMS_ALS, VAX_VMS, ANUYK44, ANAYK14, ANUYK43);

   SYSTEM_NAME  : constant NAME := VAX_VMS;

   STORAGE_UNIT : constant :=  8;
   MEMORY_SIZE  : constant :=  2**30 - 1;

   -- System-Dependent Named Numbers:

   MIN_INT      : constant := -(2**31);
   MAX_INT      : constant :=  (2**31)-1;
   MAX_DIGITS   : constant := 9;
   MAX_MANTISSA : constant := 31;
   FINE_DELTA   : constant := 2.0**(-31);
   TICK         : constant := 0.01;

   -- Other System-Dependent Declarations

   subtype PRIORITY is INTEGER range 1..15;

   --
   -- The following exceptions are provided as a "convention" whereby
   -- the Ada program can be compiled with all implicit checks suppressed
   -- (i.e., pragma SUPPRESS or equivalent), explicit checks included as
   -- necessary, the appropriate exception raised when required, and then
   -- the exception is either handled normally or the Ada program terminates.
   --
   ACCESS_CHECK         : exception;
   DISCRIMINANT_CHECK   : exception;
   INDEX_CHECK          : exception;
   LENGTH_CHECK         : exception;
   RANGE_CHECK          : exception;
   DIVISION_CHECK       : exception;
   OVERFLOW_CHECK       : exception;
   ELABORATION_CHECK    : exception;
   STORAGE_CHECK        : exception;

   --
   -- The following exceptions provide for (1) Ada programs that contain
   -- unresolved subprogram calls and (2) VAX/VMS system-level errors.
   --
   UNRESOLVED_REFERENCE : exception;
```

```
   SYSTEM_ERROR          : exception;

end SYSTEM;
```

(4) Representation Clause Restrictions

Representation clauses specify how the types of the language are to be mapped onto the underlying machine. The following are restrictions on representation clauses.

Address Clauses

No effect.

Length Clause

Length spcifications are described in Section 13.2 of the Ada Language Reference Manual.

A length specification of the form

```
T'STORAGE_SIZE for task type T;
```

specifies the number of bytes to be allocated for the run-time stack of each task object of type T.

Enumeration Representation Clause

In the absence of a representation specification for an enumeration type T, the internal representation of T'FIRST is 0. The default SIZE for a stand-alone object of enumeration type T will be the smallest of the values 8, 16, or 32, such that the internal representation of T'FIRST and T'LAST both fall within the range:

```
-2**(T'SIZE - 1) .. 2**(T'SIZE - 1)-1.
```

Length specifications of the form:

```
for T'SIZE use N;
```

and/or enumeration representations of the form:

```
for T use aggregate
```

are permitted for N in 2..32, provided the representations and the SIZE conform to the relationship specified above, or else for N in 1..31, provided that the internal

representation of T'FIRST >= 0 and the representation of T'LAST <= 2**(T'SIZE) - 1.

For components of enumeration types within packed composite objects, the smaller of the default stand-alone SIZE and the SIZE from a length specification is used.

Enumeration representation on types derived from the predefined type BOOLEAN are not accepted, but length specifications are accepted.

Array or Record Representation Clause

A length specification of the form

```
for T'SIZE use N;
```

may cause arrays and records to be packed, if required, to accommodate the length specification. If the size specified is not large enough to contain any possible value of the type, a diagnostic message of severity ERROR is generated.

The PACK pragma may be used to minimize wasted space between components of arrays and records. The pragma causes the type representation to be chosen such that storage space requirements are minimized at the possible expense of data access time and and code space.

A record type representation specification may be used to describe the allocation of components in a record. Bits are numbered 0..7 from the right. (Bit 8 starts at the right of the next higher-numbered byte.) Each location specification must allow at least X bits of range, where X is large enough to hold any value of the subtype of the component being allocated. Otherwise, a diagnostic message of severity ERROR is generated.

The alignment clause of the form:

```
at mod N
```

can specify alignment of 1 (byte), 2 (word), 4(longword).

If it is determinable at compilation time that the SIZE of a record or array type or subtype may be outside the range of STANDARD.LONG_INTEGER, a diagnostic message of severity WARNING is generated. Declaration of an object of such a type or subtype would raise NUMERIC_ERROR when elaborated. Note that a discriminated record or array may never raise the NUMERIC_ERROR when elaborated based on the actual discriminate provided.

(5) Conventions

The following conventions are used for an implementation-generated name denoting implementation-dependent components.

NONE

(6) Address Clauses

The following are conventions that define the interpretation of expressions that appear in address clauses, including those for interrupts.

NONE

(7) Unchecked Conversions

The following are restrictions on unchecked conversion, including those depending on the respective sizes of objects of the source and target.

A program is erroneous if it performs UNCHECKED-CONVERSION when the size of the source and target types have different.

(8) Input-Output Packages

The following are implementation-dependent characteristics of the input-output packages.

SEQUENTIAL_IO Package

Declare file type and applicable operations for files of this type.

The following restrictions are imposed on the use of the package SEQUENTIAL_IO:

a. A null file name parameter to the CREATE procedure (for opening a temporary file) is not appropriate, and raises the exception NAME_ERROR;

b. RESET performs no action on a character-oriented device;

c. On a disk or tape, the DELETE procedure closes the file and sets its size to zero so that its data may no longer be accessed; and

d. The subprogram END_OF_FILE always returns FALSE for a character-oriented device.

DIRECT_IO Package

The implementation-defined type COUNT that appears in Section 14.2.5 of the Ada LRM is defined as:

```
type COUNT is range 0..INTEGER'LAST;
```

TEXT_IO Package

```
-------------------------------------------------------------------------

-- PURPOSE:
--
--      TEXT_IO provides input and output services for textual files,
-- such as creation, deletion, opening, closing, reading and writing.
--
--      This package is as specified in the Ada language reference manual,
-- Section 14.

with ADA_RSL;  use ADA_RSL;
with IO_EXCEPTIONS;
package TEXT_IO is
```

```
   type FILE_TYPE is limited private;

   type FILE_MODE is (IN_FILE, OUT_FILE);

   type COUNT is range 0 .. INTEGER'LAST;
   subtype POSITIVE_COUNT is COUNT range 1 .. COUNT'LAST;
   UNBOUNDED : constant COUNT := 0;  -- line and page length

   subtype FIELD       is INTEGER range 0 .. INTEGER'LAST;
   subtype NUMBER_BASE is INTEGER range 2 .. 16;

   type TYPE_SET is (LOWER_CASE, UPPER_CASE);

   -- File Management

   procedure CREATE (FILE : in out FILE_TYPE;
                     MODE : in FILE_MODE := OUT_FILE;
                     NAME : in STRING    := "";
                     FORM : in STRING    := "");

   procedure OPEN   (FILE : in out FILE_TYPE;
                     MODE : in FILE_MODE;
                     NAME : in STRING;
                     FORM : in STRING := "");

   procedure CLOSE  (FILE : in out FILE_TYPE);
   procedure DELETE (FILE : in out FILE_TYPE);
   procedure RESET  (FILE : in out FILE_TYPE;  MODE : in FILE_MODE);
   procedure RESET  (FILE : in out FILE_TYPE);

   function  MODE   (FILE : in FILE_TYPE) return FILE_MODE;
   function  NAME   (FILE : in FILE_TYPE) return STRING;
   function  FORM   (FILE : in FILE_TYPE) return STRING;

   function IS_OPEN (FILE : in FILE_TYPE) return BOOLEAN;

   -- Control of default input and output files

   procedure SET_INPUT  (FILE : in FILE_TYPE);
   procedure SET_OUTPUT (FILE : in FILE_TYPE);

   function  STANDARD_INPUT  return FILE_TYPE;
   function  STANDARD_OUTPUT return FILE_TYPE;

   function  CURRENT_INPUT   return FILE_TYPE;
   function  CURRENT_OUTPUT  return FILE_TYPE;

   -- Specification of line and page lengths

   procedure SET_LINE_LENGTH (FILE : in FILE_TYPE; TO : in COUNT);
   procedure SET_LINE_LENGTH (TO   : in COUNT);

   procedure SET_PAGE_LENGTH (FILE : in FILE_TYPE; TO : in COUNT);
   procedure SET_PAGE_LENGTH (TO   : in COUNT);
```

```
   function  LINE_LENGTH (FILE : in FILE_TYPE) return COUNT;
   function  LINE_LENGTH  return COUNT;

   function  PAGE_LENGTH (FILE : in FILE_TYPE) return COUNT;
   function  PAGE_LENGTH  return COUNT;

   -- Column, Line, and Page Control

   procedure NEW_LINE   (FILE : in FILE_TYPE; SPACING : in POSITIVE_COUNT := 1);
   procedure NEW_LINE   (SPACING : in POSITIVE_COUNT := 1);

   procedure SKIP_LINE  (FILE : in FILE_TYPE; SPACING : in POSITIVE_COUNT := 1);
   procedure SKIP_LINE  (SPACING : in POSITIVE_COUNT := 1);

   function  END_OF_LINE (FILE : in FILE_TYPE) return boolean;
   function  END_OF_LINE  return BOOLEAN;

   procedure NEW_PAGE    (FILE : in FILE_TYPE);
   procedure NEW_PAGE;

   procedure SKIP_PAGE   (FILE : in FILE_TYPE);
   procedure SKIP_PAGE;

   function  END_OF_PAGE (FILE : in FILE_TYPE) return BOOLEAN;
   function  END_OF_PAGE  return BOOLEAN;

   function  END_OF_FILE (FILE : in FILE_TYPE) return BOOLEAN;
   function  END_OF_FILE  return BOOLEAN;

   procedure SET_COL  (FILE : in FILE_TYPE; TO : in POSITIVE_COUNT);
   procedure SET_COL  (TO   : in POSITIVE_COUNT);

   procedure SET_LINE (FILE : in FILE_TYPE; TO : in POSITIVE_COUNT);
   procedure SET_LINE (TO   : in POSITIVE_COUNT);

   function COL  (FILE : in FILE_TYPE) return POSITIVE_COUNT;
   function COL   return POSITIVE_COUNT;

   function LINE (FILE : in FILE_TYPE) return POSITIVE_COUNT;
   function LINE  return POSITIVE_COUNT;

   function PAGE (FILE : in FILE_TYPE) return POSITIVE_COUNT;
   function PAGE  return POSITIVE_COUNT;

   -- Character Input-Output

   procedure GET (FILE : in  FILE_TYPE; ITEM : out CHARACTER);
   procedure GET (ITEM : out CHARACTER);
   procedure PUT (FILE : in  FILE_TYPE; ITEM : in  CHARACTER);
   procedure PUT (ITEM : in  CHARACTER);

   -- String Input-Output

   procedure GET (FILE : in  FILE_TYPE; ITEM : out STRING);
```

```
   procedure GET (ITEM : out STRING);
   procedure PUT (FILE : in  FILE_TYPE; ITEM : in STRING);
   procedure PUT (ITEM : in  STRING);

   procedure GET_LINE (FILE : in  FILE_TYPE; ITEM : out STRING;
                       LAST : out NATURAL);
   procedure GET_LINE (ITEM : out STRING; LAST : out NATURAL);
   procedure PUT_LINE (FILE : in  FILE_TYPE; ITEM : in STRING);
   procedure PUT_LINE (ITEM : in  STRING);

   -- Generic package for Input-Output of Integer Types

   generic
      type NUM is range <>;
   package INTEGER_IO is

      DEFAULT_WIDTH : FIELD := NUM'WIDTH;
      DEFAULT_BASE  : NUMBER_BASE := 10;

      procedure GET (FILE : in  FILE_TYPE; ITEM : out NUM;
                     WIDTH : in FIELD := 0);
      procedure GET (ITEM : out NUM; WIDTH : in FIELD := 0);

      procedure PUT (FILE  : in FILE_TYPE;
                     ITEM  : in NUM;
                     WIDTH : in FIELD := DEFAULT_WIDTH;
                     BASE  : in NUMBER_BASE := DEFAULT_BASE);
      procedure PUT (ITEM  : in NUM;
                     WIDTH : in FIELD := DEFAULT_WIDTH;
                     BASE  : in NUMBER_BASE := DEFAULT_BASE);

      procedure GET (FROM : in  STRING; ITEM : out NUM; LAST : out POSITIVE);
      procedure PUT (TO   : out STRING;
                     ITEM : in  NUM;
                     BASE : in  NUMBER_BASE := DEFAULT_BASE);

   end INTEGER_IO;

   -- Generic packages for Input-Ouput of Real Types

   generic
      type NUM is digits <>;
   package FLOAT_IO is

      DEFAULT_FORE  : FIELD := 2;
      DEFAULT_AFT   : FIELD := NUM'DIGITS - 1;
      DEFAULT_EXP   : FIELD := 3;

      procedure GET (FILE : in FILE_TYPE; ITEM : out NUM;
                     WIDTH : in FIELD := 0);
      procedure GET (ITEM : out NUM; WIDTH : in FIELD := 0);

      procedure PUT (FILE : in FILE_TYPE;
                     ITEM : in NUM;
                     FORE : in FIELD := DEFAULT_FORE;
```

```
                    AFT  : in FIELD := DEFAULT_AFT;
                    EXP  : in FIELD := DEFAULT_EXP);
      procedure PUT (ITEM : in NUM;
                    FORE : in FIELD := DEFAULT_FORE;
                    AFT  : in FIELD := DEFAULT_AFT;
                    EXP  : in FIELD := DEFAULT_EXP);

      procedure GET (FROM : in STRING; ITEM : out NUM; LAST : out POSITIVE);
      procedure PUT (to   : out STRING;
                    ITEM : in NUM;
                    AFT  : in FIELD := DEFAULT_AFT;
                    EXP  : in FIELD := DEFAULT_EXP);

   end FLOAT_IO;

   generic
      type NUM is delta <>;
   package FIXED_IO is

      DEFAULT_FORE  : FIELD := NUM'FORE;
      DEFAULT_AFT   : FIELD := NUM'AFT;
      DEFAULT_EXP   : FIELD := 0;

      procedure GET (FILE : in FILE_TYPE; ITEM : out NUM;
                    WIDTH : in FIELD := 0);
      procedure GET (ITEM : out NUM; WIDTH : in FIELD := 0);

      procedure PUT (FILE : in FILE_TYPE;
                    ITEM : in NUM;
                    FORE : in FIELD := DEFAULT_FORE;
                    AFT  : in FIELD := DEFAULT_AFT;
                    EXP  : in FIELD := DEFAULT_EXP);
      procedure PUT (ITEM : in NUM;
                    FORE : in FIELD := DEFAULT_FORE;
                    AFT  : in FIELD := DEFAULT_AFT;
                    EXP  : in FIELD := DEFAULT_EXP);

      procedure GET (FROM : in STRING; ITEM : out NUM; LAST : out POSITIVE);
      procedure PUT (TO   : out STRING;
                    ITEM : in NUM;
                    AFT  : in FIELD := DEFAULT_AFT;
                    EXP  : in FIELD := DEFAULT_EXP);

   end FIXED_IO;

   -- Generic package for Input-Output of Enumeration Types

   generic
      type ENUM is (<>);
   package ENUMERATION_IO is

      DEFAULT_WIDTH    : FIELD    := 0;
      DEFAULT_SETTING  : TYPE_SET := UPPER_CASE;

      procedure GET (FILE : in FILE_TYPE; ITEM : out ENUM);
```

```
      procedure GET (ITEM : out ENUM);

      procedure PUT (FILE  : in FILE_TYPE;
                     ITEM  : in ENUM;
                     WIDTH : in FIELD    := DEFAULT_WIDTH;
                     SET   : in TYPE_SET := DEFAULT_SETTING);
      procedure PUT (ITEM  : in ENUM;
                     WIDTH : in FIELD    := DEFAULT_WIDTH;
                     SET   : in TYPE_SET := DEFAULT_SETTING);

      procedure GET (FROM : in STRING; ITEM : out ENUM; LAST : out POSITIVE);
      procedure PUT (TO   : out STRING;
                     ITEM : in  ENUM;
                     SET  : in  TYPE_SET := DEFAULT_SETTING);

   end ENUMERATION_IO;

   -- Exceptions

   STATUS_ERROR : exception renames IO_EXCEPTIONS.STATUS_ERROR;
   MODE_ERROR   : exception renames IO_EXCEPTIONS.MODE_ERROR;
   NAME_ERROR   : exception renames IO_EXCEPTIONS.NAME_ERROR;
   USE_ERROR    : exception renames IO_EXCEPTIONS.USE_ERROR;
   DEVICE_ERROR : exception renames IO_EXCEPTIONS.DEVICE_ERROR;
   END_ERROR    : exception renames IO_EXCEPTIONS.END_ERROR;
   DATA_ERROR   : exception renames IO_EXCEPTIONS.DATA_ERROR;
   LAYOUT_ERROR : exception renames IO_EXCEPTIONS.LAYOUT_ERROR;

private

   BUFFER_LENGTH : constant := 256;    -- Restrict the line length to 255
                                       -- so that each record (buffer) can
                                       -- contain a line.

   type CHAR_BUFFER is array (FILE_IO.DATA_LENGTH_INT
                              range 1 .. BUFFER_LENGTH) OF CHARACTER;


   type FILE_REC is
      -- Common file state description:  actual FILE_TYPE
      record
         STRM_PTR         : FILE_IO.STREAM_ID_PRV;
         FILES_CLASS      : FILE_IO.FILE_CLASS_ENU := FILE_IO.FC_TEXT;
         F_MODE           : FILE_MODE;
         INTERACTIVE      : BOOLEAN := FALSE;
         END_INFO         : BOOLEAN := FALSE;
         CURR_COL         : COUNT := 1;
         CURR_LINE        : COUNT := 1;
         CURR_PAGE        : COUNT := 1;
         EOLN_FOUND       : BOOLEAN := FALSE;
         EOP_FOUND        : BOOLEAN := FALSE;
         EOF_FOUND        : BOOLEAN := FALSE;
         LINE_LEN         : COUNT := UNBOUNDED;
         PAGE_LEN         : COUNT := UNBOUNDED;
         MAX_REC_LENGTH   : COUNT := BUFFER_LENGTH;
```

```
        CURR_REC_LENGTH : FILE_IO.DATA_LENGTH_INT := 0;
        TEXT_INDEX      : FILE_IO.DATA_LENGTH_INT := 0;
        TEXT_BUF        : CHAR_BUFFER;
        NEXT_REC_LENGTH : FILE_IO.DATA_LENGTH_INT := 0;
        NEXT_BUF        : CHAR_BUFFER;
        SEMAPHORE       : ADA_RSL.RESOURCE_MGR.SEMAPHORE :=
                          ADA_RSL.RESOURCE_MGR.NEW_SEMAPHORE;

    end record;

  type FILE_TYPE is access FILE_REC;

end TEXT_IO;
```

LOW_LEVEL_IO

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Include either the LOW_LEVEL_IO package specification or the
following sentence:

Low-level input-output is not provided.
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

PACKAGE STANDARD

```
------------------------------------------------------------------------

package STANDARD is

-- For this implementation, there is no cooresponding package STANDARD body.

   type BOOLEAN is (FALSE, TRUE);

-- The predefined relational operators for this type are as follows:

-- function "="  (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
-- function "/=" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
-- function "<"  (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
-- function "<=" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
-- function ">"  (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
-- function ">=" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;

-- The predefined logical operators and the predefined logical
-- negation operator are as follows:

-- function "and" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
-- function "or"  (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
-- function "xor" (LEFT, RIGHT : BOOLEAN) return BOOLEAN;

-- function "not" (RIGHT : BOOLEAN) return BOOLEAN;

------------------------------------------------------------------------
-- The universal type UNIVERSAL_INTEGER is predefined for the Ada language.
------------------------------------------------------------------------

   type INTEGER is range -32_768 .. 32_767;

-- The predefined operators for this type are as follows:

-- function "="  (LEFT, RIGHT : INTEGER) return BOOLEAN;
-- function "/=" (LEFT, RIGHT : INTEGER) return BOOLEAN;
-- function "<"  (LEFT, RIGHT : INTEGER) return BOOLEAN;
-- function "<=" (LEFT, RIGHT : INTEGER) return BOOLEAN;
-- function ">"  (LEFT, RIGHT : INTEGER) return BOOLEAN;
-- function ">=" (LEFT, RIGHT : INTEGER) return BOOLEAN;

-- function "+"   (RIGHT : INTEGER) return INTEGER;
```

```
-- function "-"   (RIGHT : INTEGER) return INTEGER;
-- function "abs" (RIGHT : INTEGER) return INTEGER;

-- function "+"   (LEFT, RIGHT : INTEGER) return INTEGER;
-- function "-"   (LEFT, RIGHT : INTEGER) return INTEGER;
-- function "*"   (LEFT, RIGHT : INTEGER) return INTEGER;
-- function "/"   (LEFT, RIGHT : INTEGER) return INTEGER;
-- function "rem" (LEFT, RIGHT : INTEGER) return INTEGER;
-- function "mod" (LEFT, RIGHT : INTEGER) return INTEGER;

-- function "**"  (LEFT : INTEGER; RIGHT : INTEGER) return INTEGER;

-- An implementation may provide additional predefined integer types.
-- It is recommended that the names of such addititional types end with
-- INTEGER as in SHORT_INTEGER or LONG_INTEGER.  The specification of
-- each operator for the type UNIVERSAL_INTEGER, or for any additional
-- predefined integer type, is obtained by replacing INTEGER by the name
-- of the type in the specification of the corresponding operator of the
-- type INTEGER, except for the right operand of the exponentiating operator.

   type LONG_INTEGER is range -2_147_483_648 .. 2_147_483_647;

-- The predefined operators for this type are as follows:

-- function "="  (LEFT, RIGHT : LONG_INTEGER) return BOOLEAN;
-- function "/=" (LEFT, RIGHT : LONG_INTEGER) return BOOLEAN;
-- function "<"  (LEFT, RIGHT : LONG_INTEGER) return BOOLEAN;
-- function "<=" (LEFT, RIGHT : LONG_INTEGER) return BOOLEAN;
-- function ">"  (LEFT, RIGHT : LONG_INTEGER) return BOOLEAN;
-- function ">=" (LEFT, RIGHT : LONG_INTEGER) return BOOLEAN;

-- function "+"   (RIGHT : LONG_INTEGER) return LONG_INTEGER;
-- function "-"   (RIGHT : LONG_INTEGER) return LONG_INTEGER;
-- function "abs" (RIGHT : LONG_INTEGER) return LONG_INTEGER;

-- function "+"   (LEFT, RIGHT : LONG_INTEGER) return LONG_INTEGER;
-- function "-"   (LEFT, RIGHT : LONG_INTEGER) return LONG_INTEGER;
-- function "*"   (LEFT, RIGHT : LONG_INTEGER) return LONG_INTEGER;
-- function "/"   (LEFT, RIGHT : LONG_INTEGER) return LONG_INTEGER;
-- function "rem" (LEFT, RIGHT : LONG_INTEGER) return LONG_INTEGER;
-- function "mod" (LEFT, RIGHT : LONG_INTEGER) return LONG_INTEGER;

-- function "**"  (LEFT : LONG_INTEGER; RIGHT : INTEGER) return LONG_INTEGER;

-------------------------------------------------------------------------
-- The universal type UNIVERSAL_REAL is predefined for the Ada language.
-------------------------------------------------------------------------

   type FLOAT is digits 6 range
            - (2#0.1111_1111_1111_1111_1111_1111#E127) ..
              (2#0.1111_1111_1111_1111_1111_1111#E127);

-- The predefined operators for this type are as follows:

-- function "="  (LEFT, RIGHT : FLOAT) return BOOLEAN;
```

```
-- function "/=" (LEFT, RIGHT : FLOAT) return BOOLEAN;
-- function "<"  (LEFT, RIGHT : FLOAT) return BOOLEAN;
-- function "<=" (LEFT, RIGHT : FLOAT) return BOOLEAN;
-- function ">"  (LEFT, RIGHT : FLOAT) return BOOLEAN;
-- function ">=" (LEFT, RIGHT : FLOAT) return BOOLEAN;

-- function "+"   (RIGHT : FLOAT) return FLOAT;
-- function "-"   (RIGHT : FLOAT) return FLOAT;
-- function "abs" (RIGHT : FLOAT) return FLOAT;

-- function "+"   (LEFT, RIGHT : FLOAT) return FLOAT;
-- function "-"   (LEFT, RIGHT : FLOAT) return FLOAT;
-- function "*"   (LEFT, RIGHT : FLOAT) return FLOAT;
-- function "/"   (LEFT, RIGHT : FLOAT) return FLOAT;

-- function "**"  (LEFT : FLOAT; RIGHT : INTEGER) return FLOAT;

-- An implemenation may provide additional predefined floating point types.
-- It is recommended that the names of such additional types end with FLOAT
-- as in SHORT_FLOAT or LONG_FLOAT.  The specification of each operator for
-- the type UNIVERSAL_REAL, or for any additional predefined floating point
-- type, is obtained by replacing FLOAT by the name of the type in the
-- specification of the corresponding opertor of the type FLOAT.

   type LONG_FLOAT is digits 9 range -
(2#0.1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111#E127)
..
(2#0.1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111#E127)
;

-- The predefined operators for this type are as follows:

-- function "="  (LEFT, RIGHT : LONG_FLOAT) return BOOLEAN;
-- function "/=" (LEFT, RIGHT : LONG_FLOAT) return BOOLEAN;
-- function "<"  (LEFT, RIGHT : LONG_FLOAT) return BOOLEAN;
-- function "<=" (LEFT, RIGHT : LONG_FLOAT) return BOOLEAN;
-- function ">"  (LEFT, RIGHT : LONG_FLOAT) return BOOLEAN;
-- function ">=" (LEFT, RIGHT : LONG_FLOAT) return BOOLEAN;

-- function "+"   (RIGHT : LONG_FLOAT) return LONG_FLOAT;
-- function "-"   (RIGHT : LONG_FLOAT) return LONG_FLOAT;
-- function "abs" (RIGHT : LONG_FLOAT) return LONG_FLOAT;

-- function "+"   (LEFT, RIGHT : LONG_FLOAT) return LONG_FLOAT;
-- function "-"   (LEFT, RIGHT : LONG_FLOAT) return LONG_FLOAT;
-- function "*"   (LEFT, RIGHT : LONG_FLOAT) return LONG_FLOAT;
-- function "/"   (LEFT, RIGHT : LONG_FLOAT) return LONG_FLOAT;

-- function "**"  (LEFT : LONG_FLOAT; RIGHT : INTEGER) return LONG_FLOAT;

-- In addition, the following operators are predefined for universal types:

-- function "*"  (LEFT : universal_integer; RIGHT : universal_real)
--                return universal_real;
-- function "*"  (LEFT : universal_real;    RIGHT : universal_integer)
```

```
--                  return universal_real;
-- function "/"   (LEFT : universal_real;     RIGHT : universal_integer)
--                  return universal_real;

-----------------------------------------------------------------------
-- The type UNIVERSAL_FIXED is predefined within the Ada language.
-----------------------------------------------------------------------

-- The only operators declared for this type are:

-- function "*"   (LEFT : any_fixed_point_type; RIGHT : any_fixed_point_type)
--                  return universal_fixed;
-- function "/"   (LEFT : any_fixed_point_type; RIGHT : any_fixed_point_type)
--                  return universal_fixed;

-----------------------------------------------------------------------
-- The following characters form the standard ASCII character set.
-----------------------------------------------------------------------

-- Character literals corresponding to control characters are not identifiers;
-- there are indicated in "italics" in this definition (i.e., unprintable).

   type CHARACTER is
      (’’, ’’, ’’, ’’,          ’’, ’’, ’’, ’’,
       ’’, ’’, ’’, ’’,
       ’’, ’’, ’’, ’’,
       ’’, ’’, ’’, ’’,
       ’ ’, ’!’, ’"’, ’#’,      ’$’, ’%’, ’&’, ’’’,
       ’(’, ’)’, ’*’, ’+’,      ’,’, ’-’, ’.’, ’/’,
       ’0’, ’1’, ’2’, ’3’,      ’4’, ’5’, ’6’, ’7’,
       ’8’, ’9’, ’:’, ’;’,      ’<’, ’=’, ’>’, ’?’,
       ’@’, ’A’, ’B’, ’C’,      ’D’, ’E’, ’F’, ’G’,
       ’H’, ’I’, ’J’, ’K’,      ’L’, ’M’, ’N’, ’O’,
       ’P’, ’Q’, ’R’, ’S’,      ’T’, ’U’, ’V’, ’W’,
       ’X’, ’Y’, ’Z’, ’[’,      ’\’, ’]’, ’^’, ’_’,
       ’‘’, ’a’, ’b’, ’c’,      ’d’, ’e’, ’f’, ’g’,
       ’h’, ’i’, ’j’, ’k’,      ’l’, ’m’, ’n’, ’o’,
       ’p’, ’q’, ’r’, ’s’,      ’t’, ’u’, ’v’, ’w’,
       ’x’, ’y’, ’z’, ’{’,      ’|’, ’}’, ’~’, ’’);
```

```
', '", '',                                  ', ',
      '', '', '', '',                       '', '', '', '',
      '', '', '$', '\', '', '', '',

      ' ', '!', '"', '#',     '$', '%', '&', ''',
      '(', ')', '*', '+',     ',', '-', '.', '/',
      '0', '1', '2', '3',     '4', '5', '6', '7',
      '8', '9', ':', ';',     '<', '=', '>', '?',

      '@', 'A', 'B', 'C',     'D', 'E', 'F', 'G',
      'H', 'I', 'J', 'K',     'L', 'M', 'N', 'O',
      'P', 'Q', 'R', 'S',     'T', 'U', 'V', 'W',
      'X', 'Y', 'Z', '[',     '\', ']', '^', '_',

      '`', 'a', 'b', 'c',     'd', 'e', 'f', 'g',
      'h', 'i', 'j', 'k',     'l', 'm', 'n', 'o',
      'p', 'q', 'r', 's',     't', 'u', 'v', 'w',
      'x', 'y', 'z', '{',     '|', '}', '~', '');

   for CHARACTER use  -- 128 ASCII character set without holes
       (  0,  1,  2,  3,  4,  5,  6,  7,  8,  9,
         10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
         20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
         30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
         40, 41, 42, 43, 44, 45, 46, 47, 48, 49,
         50, 51, 52, 53, 54, 55, 56, 57, 58, 59,
         60, 61, 62, 63, 64, 65, 66, 67, 68, 69,
         70, 71, 72, 73, 74, 75, 76, 77, 78, 79,
         80, 81, 82, 83, 84, 85, 86, 87, 88, 89,
         90, 91, 92, 93, 94, 95, 96, 97, 98, 99,
        100,101,102,103,104,105,106,107,108,109,
```

```
        110,111,112,113,114,115,116,117,118,119,
        120,121,122,123,124,125,126,127 );

-- The predefined operators for the type CHARACTER are the same as for
-- any enumeration type within the Ada language.

--------------------------------------------------------------------------
-- The package ASCII nested with the package STANDARD for the Ada language.
--------------------------------------------------------------------------

   package ASCII is

      -- Control characters:

      NUL : constant CHARACTER:='';   SOH : constant CHARACTER:='';
      STX : constant CHARACTER:='';   ETX : constant CHARACTER:='';
      EOT : constant CHARACTER:='';   ENQ : constant CHARACTER:='';
      ACK : constant CHARACTER:='';   BEL : constant CHARACTER:='';
      BS  : constant CHARACTER:='';   HT  : constant CHARACTER:='        ';
      LF  : constant CHARACTER:='
                                 '; VT  : constant CHARACTER:='
                                                              ';
      FF  : constant CHARACTER:='
```

```
';                                  '; CR  : constant CHARACTER:='
   SO  : constant CHARACTER:='';    SI  : constant CHARACTER:='';
   DLE : constant CHARACTER:='';    DC1 : constant CHARACTER:='';
   DC2 : constant CHARACTER:='';    DC3 : constant CHARACTER:='';
   DC4 : constant CHARACTER:='';    NAK : constant CHARACTER:='';
   SYN : constant CHARACTER:='';    ETB : constant CHARACTER:='';
   CAN : constant CHARACTER:='';    EM  : constant CHARACTER:='';
   SUB : constant CHARACTER:='?';   ESC : constant CHARACTER:='
   FS  : constant CHARACTER:='^\';   GS  : constant CHARACTER:='';
   RS  : constant CHARACTER:='';    US  : constant CHARACTER:='';
   DEL : constant CHARACTER:='';

   -- Other characters

   EXCLAM     : constant CHARACTER:='!'; QUOTATION  : constant CHARACTER:='"';
   SHARP      : constant CHARACTER:='#'; DOLLAR     : constant CHARACTER:='$';
   PERCENT    : constant CHARACTER:='%'; AMPERSAND  : constant CHARACTER:='&';
   COLON      : constant CHARACTER:=':'; SEMICOLON  : constant CHARACTER:=';';
   QUERY      : constant CHARACTER:='?'; AT_SIGN    : constant CHARACTER:='@';
   L_BRACKET  : constant CHARACTER:='['; BACK_SLASH : constant CHARACTER:='\';
   R_BRACKET  : constant CHARACTER:=']'; CIRCUMFLEX : constant CHARACTER:='^';
   UNDERLINE  : constant CHARACTER:='_'; GRAVE      : constant CHARACTER:='`';
   L_BRACE    : constant CHARACTER:='{'; BAR        : constant CHARACTER:='|';
   R_BRACE    : constant CHARACTER:='}'; TILDE      : constant CHARACTER:='~';

   -- Lower case letters (PRINTABLE?)-- Upper case letters (PRINTABLE!)

   LC_A : constant CHARACTER:='a';    -- UC_A : constant CHARACTER:='A';
   LC_B : constant CHARACTER:='b';    -- UC_B : constant CHARACTER:='B';
   LC_C : constant CHARACTER:='c';    -- UC_C : constant CHARACTER:='C';
   LC_D : constant CHARACTER:='d';    -- UC_D : constant CHARACTER:='D';
   LC_E : constant CHARACTER:='e';    -- UC_E : constant CHARACTER:='E';
   LC_F : constant CHARACTER:='f';    -- UC_F : constant CHARACTER:='F';
   LC_G : constant CHARACTER:='g';    -- UC_G : constant CHARACTER:='G';
   LC_H : constant CHARACTER:='h';    -- UC_H : constant CHARACTER:='H';
```

```
      LC_I : constant CHARACTER:='i';    -- UC_I : constant CHARACTER:='I';
      LC_J : constant CHARACTER:='j';    -- UC_J : constant CHARACTER:='J';
      LC_K : constant CHARACTER:='k';    -- UC_K : constant CHARACTER:='K';
      LC_L : constant CHARACTER:='l';    -- UC_L : constant CHARACTER:='L';
      LC_M : constant CHARACTER:='m';    -- UC_M : constant CHARACTER:='M';
      LC_N : constant CHARACTER:='n';    -- UC_N : constant CHARACTER:='N';
      LC_O : constant CHARACTER:='o';    -- UC_O : constant CHARACTER:='O';
      LC_P : constant CHARACTER:='p';    -- UC_P : constant CHARACTER:='P';
      LC_Q : constant CHARACTER:='q';    -- UC_Q : constant CHARACTER:='Q';
      LC_R : constant CHARACTER:='r';    -- UC_R : constant CHARACTER:='R';
      LC_S : constant CHARACTER:='s';    -- UC_S : constant CHARACTER:='S';
      LC_T : constant CHARACTER:='t';    -- UC_T : constant CHARACTER:='T';
      LC_U : constant CHARACTER:='u';    -- UC_U : constant CHARACTER:='U';
      LC_V : constant CHARACTER:='v';    -- UC_V : constant CHARACTER:='V';
      LC_W : constant CHARACTER:='w';    -- UC_W : constant CHARACTER:='W';
      LC_X : constant CHARACTER:='x';    -- UC_X : constant CHARACTER:='X';
      LC_Y : constant CHARACTER:='y';    -- UC_Y : constant CHARACTER:='Y';
      LC_Z : constant CHARACTER:='z';    -- UC_Z : constant CHARACTER:='Z';

   end ASCII;

-------------------------------------------------------------------------------
-- Predefined subtypes within the Ada language:
-------------------------------------------------------------------------------

   subtype NATURAL  is INTEGER range 0 .. INTEGER'LAST;
   subtype POSITIVE is INTEGER range 1 .. INTEGER'LAST;

--
-- with the LONG_INTEGER type defined, these subtypes follow
--
   subtype LONG_NATURAL  is LONG_INTEGER range 0 .. LONG_INTEGER'LAST;
   subtype LONG_POSITIVE is LONG_INTEGER range 1 .. LONG_INTEGER'LAST;

-------------------------------------------------------------------------------
-- Predefined STRING type within the Ada language:
-------------------------------------------------------------------------------

   type  STRING   is array (POSITIVE range <>) of CHARACTER;

   pragma PACK(STRING);

-- The predefined operators for this type are as follows:

-- function "="    (LEFT, RIGHT : STRING) return BOOLEAN;
-- function "/="   (LEFT, RIGHT : STRING) return BOOLEAN;
-- function "<"    (LEFT, RIGHT : STRING) return BOOLEAN;
-- function "<="   (LEFT, RIGHT : STRING) return BOOLEAN;
-- function ">"    (LEFT, RIGHT : STRING) return BOOLEAN;
-- function ">="   (LEFT, RIGHT : STRING) return BOOLEAN;

-- function "&"   (LEFT : STRING;    RIGHT : STRING)    return STRING;
-- function "&"   (LEFT : CHARACTER; RIGHT : STRING)    return STRING;
-- function "&"   (LEFT : STRING;    RIGHT : CHARACTER) return STRING;
-- function "&"   (LEFT : CHARACTER; RIGHT : CHARACTER) return STRING;
```

```
--------------------------------------------------------------------------
-- The type DURATION is predefined for use with Ada DELAY statement.
--------------------------------------------------------------------------

   type DURATION is delta 2.0 ** (-14)
                         range -131_072.0 .. 131_072.0 - 2.0 ** (-14);

-- The predefined operators for the type DURATION are the same as for
-- any fixed point type within the Ada language.

--------------------------------------------------------------------------
-- The predefined exceptions within the Ada language:
--------------------------------------------------------------------------

   CONSTRAINT_ERROR : exception;
   NUMERIC_ERROR    : exception;
   PROGRAM_ERROR    : exception;
   STORAGE_ERROR    : exception;
   TASKING_ERROR    : exception;

end STANDARD;
```

(10) File names

File names follow the conventions and restrictions of {the target operating system | Ada identifiers | ALS/N_Node naming conventions}.

APPENDIX C

TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below.

Name and Meaning	Value
$BIG_ID1 Identifier the size of the maximum input line length with varying last character.	<1..119 => 'A', 120 => '1'>
$BIG_ID2 Identifier the size of the maximum input line length with varying last character.	<1..119 => 'A', 120 => '2'>
$BIG_ID3 Identifier the size of the maximum input line length with varying middle character.	<1..69 => 'A', 70 => '3', 71..120 => 'A'>
$BIG_ID4 Identifier the size of the maximum input line length with varying middle character.	<1..69 => 'A', 70 => '4', 71..120 => 'A'>
$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	<1..117 => '0', 118..120 => '298'>
$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	<1..115 => '0', 116..120 => '690.0'>

$BIG_STRING1 A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.	<1 => '"', 2..61 => 'A', 62 => '"'>
$BIG_STRING2 A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.	<1 => '"', 2..60 => 'A', 61 => '1', 62 => '"'>
$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	<1..100 => ' '>
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	32767
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	32767
$FILE_NAME_WITH_BAD_CHARS An external file name that either contains invalid characters or is too long.	BAD_CHARS #.%!X
$FILE_NAME_WITH_WILD_CARD_CHAR An external file name that either contains a wild card character or is too long.	WILD_CHAR*.NAM
$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	75_000.0
$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	131_073.0
$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	BADCHAR @. !

$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	MUCH_TOO_LONG_NAME_FOR_A_FILE_UNDER_VMS_SO_THERE
$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-32768
$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	32767
$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	32768
$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-75_000.0
$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-131_073.0
$MAX_DIGITS Maximum digits supported for floating-point types.	9
$MAX_IN_LEN Maximum input line length permitted by the implementation.	120
$MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	2147483647
$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	2147483648
$MAX_LEN_INT_BASED_LITERAL A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	<1..2 => '2:', 3..117 => '0', 118..120 => '11:'>

$MAX_LEN_REAL_BASED_LITERAL A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	<1..3 => '16:', 4..116 => '0', 117..120 => 'F.E:'>
$MAX_STRING_LITERAL A string literal of size MAX_IN_LEN, including the quote characters.	<1 => '"', 2..119 => 'A', 120 => '"'>
$MIN_INT A universal integer literal whose value is SYSTEM.MIN_ INT.	-2147483648
$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	No_Such_Type
$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	16#FFFFFFFE#

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 28 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form "AI-ddddd" is to an Ada Commentary.

B28003A: A basic declaration (line 36) wrongly follows a later declaration.

E28005C: This test requires that 'PRAGMA LIST (ON);' not appear in a listing that has been suspended by a previous "pragma LIST (OFF);"; the Ada Standard is not clear on this point, and the matter will be reviewed by the ARG.

C34004A: The expression in line 168 wrongly yields a value outside of the range of the target type T, raising CONSTRAINT_ERROR.

C35502P: Equality operators in lines 62 & 69 should be inequality operators.

A35902C: Line 17's assignment of the nominal upper bound of a fixed-point type to an object of that type raises CONSTRAINT_ERROR, for that value lies outside of the actual range of the type.

C35904A: The elaboration of the fixed-point subtype on line 28 wrongly raises CONSTRAINT_ERROR, because its upper bound exceeds that of the type.

C35904B: The subtype declaration that is expected to raise CONSTRAINT_ERROR when its compatibility is checked against that of various types passed as actual generic parameters, may in fact raise NUMERIC_ERROR or CONSTRAINT_ERROR for reasons not anticipated by the test.

C35A03E, & R: These tests assume that attribute 'MANTISSA returns 0 when applied to a fixed-point type with a null range, but the Ada Standard doesn't support this assumption.

C37213H: The subtype declaration of SCONS in line 100 is wrongly expected to raise an exception when elaborated.

C37213J: The aggregate in line 451 wrongly raises CONSTRAINT_ERROR.

C37215C, E, G, H: Various discriminant constraints are wrongly expected to be incompatible with type CONS.

C38102C: The fixed-point conversion on line 23 wrongly raises CONSTRAINT_ERROR.

C41402A: 'STORAGE_SIZE is wrongly applied to an object of an access type.

C45332A: The test expects that either an expression in line 52 will raise an exception or else MACHINE_OVERFLOWS is FALSE. However, an implementation may evaluate the expression correctly using a type with a wider range than the base type of the operands, and MACHINE_OVERFLOWS may still be TRUE.

C45614C: REPORT.IDENT_INT has an argument of the wrong type (LONG_INTEGER).

E66001D: Wrongly allows either the acceptance or rejection of a parameterless function with the same identifier as an enumeration literal; the function must be rejected (see Commentary AI-00330).

A74106C, C85018B, C87B04B, CC1311B: A bound specified in a fixed-point subtype declaration lies outside of that calculated for the base type, raising CONSTRAINT_ERROR. Errors of this sort occur re lines 37 & 59, 142 & 143, 16 & 48, and 252 & 253 of the four tests, respectively (and possibly elsewhere).

BC3105A: Lines 159..168 are wrongly expected to be illegal; they are legal.

AD1A01A: The declaration of subtype INT3 raises CONSTRAINT_ERROR for implementations that select INT'SIZE to be 16 or greater.

CE2401H: The record aggregates in lines 105 & 117 contain the wrong values.

CE3208A: This test expects that an attempt to open the default output file (after it was closed) with mode IN_FILE raises NAME_ERROR or USE_ERROR; by Commentary AI-00048, MODE_ERROR should be raised.